Posebne metode ali Kako delujejo razredi

Konstruktor je metoda, ki jo Python pokliče takoj po tem, ko sestavi objekt. Sam od sebe, ne da bi jo morali klicati ročno. Da bi Python vedel, da gre za konstruktor, pa mu moramo dati vnaprej določeno ime `__init__`. Konstruktorja nikoli ne kličemo sami: naša naloga je le, da ga definiramo (če ga razred potrebuje), klical pa ga bo Python.

Poleg konstruktorja obstajajo še druge *posebne metode* z vnaprej določenimi imeni, ki se obnašajo ravno tako: mi jih le definiramo, Python pa jih kliče. Te metode razredu dajejo potrebne operatorje in funkcionalnosti.

Definirajmo razred `Vector`.

```
class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)
```

Razred deluje s poljubno dimenzionalnimi vektorji. Konstruktor sprejme toliko argumentov, kolikor mu jih damo. Znašli se bodo v terki `coords`, ki jo lepo shranimo v `self.coords`. S tem, ko rečemo `self.coords = list(coords)` poskrbimo, da bo `self.coords` seznam in ne terka. (To nam bo prišlo prav kasneje.)

Sestavimo en tak vektor in ga izpišimo.

```
v = Vector(3, 4, 1)
```

```
v
```

```
<__main__.Vector at 0x7fd3f3867790>
```

```
print(v)
```

```
<__main__.Vector object at 0x7fd3f3867790>
```

Hm, `<__main__.Vector at 0x102e29358>`? A ne bi bilo lepše, če bi se vektor izpisal kot, recimo `<3, 4, 1>`? Kako bi prepričali Python, naj sestavi lepši izpis tega vektorja?

Izpis

Python v resnici ne sestavlja izpisov. Python reče objektu, naj sestavi primeren izpis samega sebe in ga vrne kot niz. To mu "reče" tako, da pokliče njegovo metodo `__str__`. (Pa če je nima? Ima jo. Vsi razredi so izpeljani iz "prarazreda" `object` in ta ima metodo `__str__`, ki sestavi izpis, ki ga vidimo zgoraj, `<__main__.Vector at 0x102e29358>`, ki je sestavljen iz imena razreda in pomnilniškega naslova, kjer je ta objekt shranjen.)

Razredu torej dodajmo metodo `__str__`, ki vrne niz s primerno predstavitvijo tega objekta.

Še enkrat: `__str__` ne izpisuje, temveč vrne predstavitev objekta v obliki niza. "Pretvarja v niz", če hočete.

```
class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"
```

Spredaj in zadaj bosta `<` in `>`, vmes pa z vejicami združimo vse številke v `self.coords`.

```
print(v)
```

```
<__main__.Vector object at 0x7fd3f3867790>
```

Isto?

Seveda. Razredi so dinamične zadeve. Tule smo definirali nov razred `Vector`, `v` pa je še vedno primerek starega razreda. Če hočemo tak objekt `Vector`, ki se bo znal tudi izpisovati, ga bo potrebno sestaviti na novo.

```
v = Vector(3, 4, 1)

print(v)
```

```
<3, 4, 1>
```

Še vedno. Kaj pa brez `print`?

```
v
```

```
<__main__.Vector at 0x7fd3f386ac10>
```

Tu gre za dva različna opisa objektov. Izpis s `print` pokaže "prijazno predstavitev". Izpis brez `print`-a (ki ga vidimo predvsem v ukazni vrstici) pa pokaže predstavitev, ki je, če se da, v takšni obliki, da bi jo lahko le skopirali nazaj v ukazno vrstico in dobili nov takšen objekt.

To se najbolj vidi pri nizih.

```
s = "Benjamin"

print(s)
```

```
Benjamin
```

```
s
```

```
'Benjamin'
```

S `print` izpišemo niz, brez tega pa dobimo niz z narekovaji - tako, da bi lahko točno to besedilo skopirali nazaj v izraz, recimo, `ime = ...`.

Izpisa gresta prek različnih metod. `print` gre prek `__str__`, izpis v ukazno vrstico pa predk `__repr__`.

Pa naredimo podobno še z vektorjem. Metoda, ki jo Python pokliče, ko hoče izpisati objekt na ta način, je `__repr__` (*represent*).

```
class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"
```

```
v = Vector(3, 4, 1)
```

```
print(v)
```

```
<3, 4, 1>
```

```
v
```

```
Vector(3, 4, 1)
```

Mimogrede, kako pridemo do tega izpisa, če nismo v ukazni vrstici? Predvsem do slednjega?

Se spomnite funkcije `str`, ki pretvori karkoli (recimo število) v niz? Seveda se je spomnimo, saj jo prav zgoraj tudi uporabljamo. Funkcija ne zna pretvarjati v nize le številk, temveč karkoli.

```
a = 42
str(a)
```

```
'42'
```

```
e = [1, 2, 3]
str(e)
```

```
'[1, 2, 3]'
```

```
d = {1: 4, "Ana": None}
str(d)
```

```
"{1: 4, 'Ana': None}"
```

Zna figo. Funkcija ne dela ničesar, le objektu reče, naj se opiše z nizom. `str` v resnici pokliče objektovo metodo `__str__`. Če imamo neko reč `o` in napišemo `str(o)`, v resnici dobimo, kar vrne `o.__str__()`.

```
a.__str__()
```

```
'42'
```

```
e.__str__()
```

```
'[1, 2, 3]'
```

```
d.__str__()
```

```
"{1: 4, 'Ana': None}"
```

Vse delo torej opravijo objekti. Zato pa imamo objektno programiranje. V pravih objektnih jezikih objekti poskrbijo za vse, vključno s svojim izpisom.

Za še bolj direkten dokaz:

```
(42).__str__()
```

```
'42'
```

```
[1, 2, 3].__str__()
```

```
'[1, 2, 3]'
```

```
{1: 4, "Ana": None}.__str__()
```

```
"{1: 4, 'Ana': None}"
```

(Zakaj sem stisnil `42` v oklepaje? Ker bi Python ob piki sicer pomislil, da mu podajam decimalke.)

Tako kot druge stvari, zna `str` pretvarjati v nize tudi naše vektorje.

```
str(v)
```

```
'<3, 4, 1>'
```

Poleg `str` imamo tudi `repr`, ki kliče `__repr__`. Včasih sta enaka, včasih ne.

```
repr(42)
```

```
'42'
```

```
repr("Benjamin")
```

```
"'Benjamin'"
```

```
repr(v)
```

```
'Vector(3, 4, 1)'
```

Zanimivo je predvsem, kar je vrnil z nizom: niz, ki vsebuje opis niza, torej z oklepaji.

Še enkrat: metod `__str__` in `__repr__` nikoli ne kličemo direktno. Kličejo se ob različnih priložnostih (npr. `print`), "direktno" pa jih kličemo s `str` in `repr`. Podoben vzorec bomo videli tudi druge.

Zakaj tako? Zato, ker funkcije, kot so `str` in `repr` včasih še kaj dodajo. Včasih kaj preverijo, včasih pa znajo takrat, ko ustrezna posebna metoda na obstaja, poiskati kakšno rezervno pot.

```
class A:
    def __str__(self):
        return 42

a = A()

a.__str__()
```

```
42
```

```
str(a)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-bddfa438ffc2> in <module>
----> 1 str(a)

TypeError: __str__ returned non-string (type int)
```

Dolžina, velikost

Zdaj pa želimo dobiti velikost vektorja - v smislu dimenzionalnosti. Nekaj takega:

```
len(v)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-a878aa1e2fde> in <module>
----> 1 len(v)

TypeError: object of type 'Vector' has no len()
```

Python spet lenari. Glejte, kaj nam je odgovoril: ni rekel, da ne more povedati dolžine vektorja `v`, temveč pravi, da objekti vrste `Vector` nimajo dolžine. Očitno spet namerava narediti isto, kar počne že ves čas: vprašati vektor, kako dolg je.

Če torej hočemo, da bo imel vektor dolžino, moramo napisati primerno metodo. Imenuje se - kdo bi si mislil! - `__len__`.

```
class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"
```

```
    def __len__(self):
        return len(self.coords)
```

```
v = Vector(3, 4, 1)
```

```
len(v)
```

```
3
```

Zdaj pa še prava dolžina ali, kot bi rekli matematiki, absolutna vrednost. Radi bi, da bi `abs(v)` namesto

```
abs(v)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-09d4e6f30397> in <module>
----> 1 abs(v)

TypeError: bad operand type for abs(): 'Vector'
```

vrnil "absolutno vrednost* vektorja.

```
from math import sqrt

class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"

    def __len__(self):
        return len(self.coords)

    def __abs__(self):
        return sqrt(sum(x ** 2 for x in self.coords))
```

Preskusimo.

```
v = Vector(3, 4, 1)
```

```
abs(v)
```

```
5.0990195135927845
```

Prave operacije z vektorji

Igra je torej očitna: Pythonove funkcije ne delajo ničesar. Za čisto vsako reč pokličejo metode objektov, ki mu priskrbijo določeno funkcionalnost. Če te metode obstajajo. Če ne, pač ne in Python bo rekel, da se to ne da.

Vzemimo, recimo, seštevanje.

```
v = Vector(3, 4, 1)
u = Vector(-2, 3, 0)
v + u
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-c2f2f0001713> in <module>
      1 v = Vector(3, 4, 1)
      2 u = Vector(-2, 3, 0)
----> 3 v + u

TypeError: unsupported operand type(s) for +: 'Vector' and 'Vector'
```

Metoda, ki jo Python pokliče, ko hočemo kaj sešteti, se imenuje `__add__`. Dobila bo dva argumenta: prvi bo, kot običajno `self`, drugi pa bo vektor, ki ga želimo prišteti k temu vektorju.

Aha, torej nekaj takega?

```
from math import sqrt

class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"

    def __len__(self):
        return len(self.coords)

    def __abs__(self):
        return sqrt(sum(x ** 2 for x in self.coords))

    def __add__(self, other):
        return [x + y for x, y in zip(self.coords, other.coords)]
```

Hmnja. Nekaj takega, ja.

```
v = Vector(3, 4, 1)
u = Vector(-2, 3, 0)
v + u
```

```
[1, 7, 1]
```

Deluje. Ko smo računali `v + u`, je Python v resnici poklical našo metodo `__add__`. Vendar ne vrača čisto tega, kar bi hoteli. Vrne seznam. Vsota dveh vektorjev pa bi morala biti vektor.

S tem je pač tako: metoda vrača, kar vrača. Pri nekaterih smo omejeni: metoda `__str__` *mora* vrniti niz, sicer se bo Python pritožil.

Razultat seštevanja pa je lahko karkoli. Če se nam zdi dobra ideja definirati `__add__` kot

```
    def __add__(self, other):
        return "Benjamin"
```

bo `v + u` pač `"Benjamin"`.

Če hočemo, da bi bila vsota vektorjev vektor, pa bo morala `__add__` pač vračati vektor.

```
from math import sqrt

class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"

    def __len__(self):
        return len(self.coords)

    def __abs__(self):
        return sqrt(sum(x ** 2 for x in self.coords))

    def __add__(self, other):
        return Vector(*(x + y for x, y in zip(self.coords, other.coords)))
```

Metoda je podobna kot prej, le da s tem seznamom zdaj pokličemo `Vector`, da tako dobimo nov vektor.

```
v = Vector(3, 4, 1)
u = Vector(-2, 3, 0)
u + v
```

```
Vector(1, 7, 1)
```

Ne spreglejte zvezdice v klicu: s tem dosežemo, da so vsi elementi seznama argumenti za "funkcijo" `Vector`. Seznam se razpakira v argumente klica.

Kaj pa množenje vektorjev? Hm, množenje - s čim? Vektor lahko pomnožimo s skalarjem (po domače, s številko, recimo `u * 3`) ali z drugim vektorjem. Ko množimo z drugim vektorjem je produkt lahko skalarni ali kak drugačen. Ker tule delamo s splošnimi vektorji (čeprav imamo v primeru zgolj slučajno ravno tridimenzionalne), se bomo dogovorili, da bodo naši produkti skalarni.

Metoda, ki jo potrebujemo, se imenuje `__mul__`. Ali množimo s skalarjem ali vektorjem, pa bomo preverili v sami metodi: če je drugi argument vektor, množimo z vektorjem, sicer vsako kompomento posebej pomnožimo s skalarjem.

Smo že kdaj videli funkcijo `isinstance`? Podamo ji nek objekt in tip; vrne `True`, če je podani objekt podanega tipa (ali pa izpeljan iz njega).

```
isinstance("Benjamin", str)
```

```
True
```

```
isinstance(u, str)
```

```
False
```

```
isinstance(u, Vector)
```

```
True
```

Oboroženi z `isinstance` brez težav spišemo `__mul__`.

```
from math import sqrt

class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"

    def __len__(self):
        return len(self.coords)

    def __abs__(self):
        return sqrt(sum(x ** 2 for x in self.coords))

    def __add__(self, other):
        return Vector(*(x + y for x, y in zip(self.coords, other.coords)))
```

```
    def __mul__(self, other):
        if isinstance(other, Vector):
            if len(self) != len(other):
                raise ValueError("vector dimensionality mismatch")
            return sum(x * y for x, y in zip(self.coords, other.coords))
        else:
            return Vector(*(x * other for x in self.coords))
```

Skalarni produkt:

```
v = Vector(3, 4, 1)
u = Vector(-1, 3, 0)
v * u
```

```
9
```

In produkt s skalarjem:

```
v * 3
```

```
Vector(9, 12, 3)
```

Skalar ni nujno celo število. Seveda ne, tudi metoda ne predpostavlja ničesar o tem, kakšnega tipa je `other`. Če ni `Vector`, ga množi. Zato smemo seveda množiti tudi z 2.5.

```
v * 2.5
```

```
Vector(7.5, 10.0, 2.5)
```

Hm, pa z nizom?

```
v * "x"
```

```
Vector(xxx, xxxx, x)
```

To deluje, ker Python lahko množi cela števila in nize. "Vektor", ki smo ga dobili, pa je nesmiselen. Nekatere operacije celo podpira: prištejemo mu lahko drug takšen vektor iz nizov. Če poskusimo izračunati absolutno vrednost takega "vektorja", pa se bo Python pritožil, da ne zna kvadrirati nizov.

```
a = Vector("Ana", "Dani")
b = Vector("marija", "ela")
a + b
```

```
Vector(Anamarija, Daniela)
```

Python je pač liberalen jezik. Nobenih predpostavk o tem, kaj je smiselno in kaj želimo, ne dela. Pusti nam narediti vse. Laissez-faire. Seveda pa je naš problem, če to svobodo izrabimo in počnemo kozlarije.

Vrnimo se na trda tla matematike. Vektorjev navadno ne množimo s skalarjem z desne, temveč z leve. Razlike sicer ni, gre samo za stvar zapisa. Obrnimo torej.

```
3 * v
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-5446377422ad> in <module>
----> 1 3 * v

TypeError: unsupported operand type(s) for *: 'int' and 'Vector'
```

Zakaj pa to ne dela?!

`Vector`-jeva metoda `__mul__(self, other)` ve, kako pomnožiti `self` z `other`, ne pa tudi, kako pomnožiti `other` s `self`. A ni to isto? Ne, ne nujno. Kot prvo, to ne drži za vse operacije. Za množenje že, za odštevanje pa `self - other` ni prav zelo isto kot `other - self`. Kot drugo, tudi množenje ni vedno komutativno. Ko, recimo, množimo matrike, `A * B` ni isto kot `B * A`; če nimamo ravno sreče, se lahko zgodi celo, da se enega od teh produktov sploh ne da izračunati.

Skratka, Python je liberalen, ni pa naiven. Ne poenostavlja, kjer se morda ne sme.

Kako pa bomo potem prepričali Python, da nam bo vseeno izračunal, koliko je `3 * v`? Očitno bi za tole moral poskrbeti `__mul__`, ki ga ima `int`.

Ima tudi `int` metodo `__mul__`. Seveda. Ste mar mislili, da zna Python poštevanko? Niti slučajno! Tudi ko Python množi števila, v resnici kliče `__mul__`.

```
a = 6
b = 7
a * b
```

```
42
```

Tule Python seveda v resnici pokliče

```
a.__mul__(b)
```

```
42
```

Če kliče `__mul__` za množenje vektorjev, kliče `__mul__` tudi za množenje števil. Števila (`int`) v Pythonu niso nobena izjema, z njimi ravna kot z vsemi drugimi podatkovnimi tipi.

Kakorkoli že, `int` ima metodo `__mul__`, ki pa kot drugi argument (`other`) ne sprejema `Vector`. Iz dveh razlogov. Prvi je, da tisti, ki je programiral `int` ni mogel pričakovati, da bomo nekoč definirali nek `Vector`, s katerim si bomo želeli množiti njegove `int`e. Drugi je, da je stvari, ki bi jih želeli množiti s skalarji, še polno. Recimo nizi in seznami. In tudi tam lahko pišemo `"Ana" * 3` in `3 * "Ana"`. Kdo poskrbi za tidve množenji? Nizi. Številke so bolj osnovne.

Tudi za množenje vektorjev bomo morali poskrbeti sami. Vendar ne z `__mul__`. Obstaja druga metoda, `__rmul__`, ki sicer ravno tako sprejme dva argumenta,

ki je bomo ravno tako imenovali `self` in `other`, vendar se razume, da je `self` na desno strani množenja.

Videti metoda `__rmul__`? Pravzaprav enako. Potem pa kar povejmo, da je pravzaprav ista, saj je pri množenju vektorjev s skalarji vseeno, kdo je na levi in kdo na desni.

```
from math import sqrt

class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"

    def __len__(self):
        return len(self.coords)

    def __abs__(self):
        return sqrt(sum(x ** 2 for x in self.coords))

    def __add__(self, other):
        return Vector(*(x + y for x, y in zip(self.coords, other.coords)))

    def __mul__(self, other):
        if isinstance(other, Vector):
            if len(self) != len(other):
                raise ValueError("vector dimensionality mismatch")
            return sum(x * y for x, y in zip(self.coords, other.coords))
        else:
            return Vector(*(x * other for x in self.coords))

    __rmul__ = __mul__
```

Hm, a tole ... res deluje? Prirejanje znotraj razreda? Zakaj pa ne? Znotraj `class` pač pišemo funkcije, prirejamo vrednosti (razrednim) spremenljivkam ...

```
v = Vector(3, 4, 1)
```

```
3 * v
```

```
Vector(9, 12, 3)
```

Dodamo še odštevanje in deljenje?

```
from math import sqrt

class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"

    def __len__(self):
        return len(self.coords)

    def __abs__(self):
        return sqrt(sum(x ** 2 for x in self.coords))

    def __neg__(self):
        return -1 * self

    def __add__(self, other):
        return Vector(*(x + y for x, y in zip(self.coords, other.coords)))

    def __sub__(self, other):
        return self + -other

    def __mul__(self, other):
        if isinstance(other, Vector):
            if len(self) != len(other):
                raise ValueError("vector dimensionality mismatch")
            return sum(x * y for x, y in zip(self.coords, other.coords))
        else:
            return Vector(*(x * other for x in self.coords))

    __rmul__ = __mul__

    def __truediv__(self, other):
        return self * (1 / other)
```

Začeli smo z negacijo, `__neg__`. Z njo dosežemo, da bo Python znal izračunati nasprotno vrednost, `-v`.

```
v = Vector(3, 4, 1)
u = Vector(-2, 3, 0)
-u
```

```
Vector(2, -3, 0)
```

Python ima dva operatorja `-` - unarnega (negacijo) in binarnega (odštevanje). Z `__neg__` definiramo binarnega.

`__neg__` smo definirali preprosto kot množenje z `-1`. Ker pač znamo množiti vektorje s skalarji in kar matematiki pravijo, da je aditivni inverz vektorja enak množenju z nasprotno vrednostjo enote skalarja (upam, da je to res res :), pač množimo z -1.

S tem zdaj lahko sprogramiramo odštevanje z leve in z desne: odštevanje ni nič drugega kot prištevanje nasprotne vrednosti. Napisali smo samo `__sub__`; `__rsub__` ne potrebujemo, saj bomo vedno odštevali dva vektorja, torej bo vedno obstajal in deloval že `__sub__`.

Končno še deljenje. Metoda se imenuje `__truediv__` - za razliko od `__floordiv__`, ki predstavlja celoštevilsko deljenje. Prvemu ustreza `/`, drugemu pa `//`. Deljenja z leve ne bomo implementirali, saj `3 / v` ne obstaja.

Dostop do komponent

Vse, kar nam še manjka, je dostop do komponent tega našega vektorja. Lepo bi bilo, če bi lahko rekli `v[1]`. Trenutno do prve komponente pridemo z `v.coords[1]`, kar pa ni ravno najlepše.

Ko že vse tole predavanje razlagam, kako Python ne zna ničesar - ja, tako daleč gre to. Če imam nek seznam `s` in napišem `s[5]`, ni Python tisti, ki vrne peti element seznama. Ne, Python reče seznamu, da hoče dobiti peti element. Res, Python čisto ničesar ne dela sam; vsako stvar prepusti objektom, s katerimi dela. (No, da ne pretiravam: Python dela vse, kar se ne tiče objektov. Python skrbi za lokalne in globalne spremenljivke, prireja vrednosti imenom, skrbi za čiščenje pomnilnika... Je pa bolj "organizator", medtem ko delajo drugi. Poleg tega se moramo zmeniti za definicijo tega, čemu rečemo "Python".)

Metoda, ki vrača elemente, je `__getitem__`; kot argument dobi, poleg `self`, indeks elementa. Metoda, ki jih nastavlja, je `__setitem__`.

```
from math import sqrt

class Vector:
    def __init__(self, *coords):
        self.coords = list(coords)

    def __str__(self):
        return f"<{', '.join(str(coords) for coords in self.coords)}>"

    def __repr__(self):
        return f"Vector({', '.join(str(coord) for coord in self.coords)})"
```

```
    def __len__(self):
        return len(self.coords)

    def __abs__(self):
        return sqrt(sum(x ** 2 for x in self.coords))

    def __neg__(self):
        return -1 * self

    def __add__(self, other):
        return Vector(*(x + y for x, y in zip(self.coords, other.coords)))

    def __sub__(self, other):
        return self + -other

    def __mul__(self, other):
        if isinstance(other, Vector):
            if len(self) != len(other):
                raise ValueError("vector dimensionality mismatch")
            return sum(x * y for x, y in zip(self.coords, other.coords))
        else:
            return Vector(*(x * other for x in self.coords))

    __rmul__ = __mul__

    def __truediv__(self, other):
        return self * (1 / other)

    def __getitem__(self, i):
        return self.coords[i]

    def __setitem__(self, i, coord):
        self.coords[i] = coord
```

Zdaj `Vector` podpira indeksiranje.

```
v = Vector(3, 4, 1)
v[1]
```

```
4
```

```
v[2] = 13
v
```

```
Vector(3, 4, 13)
```

Končni rezultat

Dobili smo razred za vektorje, s katerim lahko računamo, kot se pač računa z vektorji.

Sestavimo tri vektorje.

```
a = Vector(4, 1, 3)
b = Vector(-1, 7, 3)
c = Vector(1, 0, 3)
```

Izračunajmo kaj z njimi.

```
2 * (a + b) - c
```

```
Vector(5, 16, 9)
```

Kakšen pa je skalarni produkt te reči z `a`?

```
(2 * (a + b) - c) * a
```

```
63
```

Vse deluje, kot mora.

Kaj bi še lahko dodali?

Bi lahko vektorjem dali še kaj? Lahko bi, recimo, rekli, da so ničelni vektorji neresnični.

```
    def __bool__(self):
        return any(self.values)
```

Vendar ustavimo konje.

Bi lahko dodali kak svoj operator, recimo `a $ b`, ki bi pomeni, na primer, vektorski produkt? Ne. Pišemo lahko le metode za obstoječe operatorje. Če Python ne podpira izraza `a $ b` (in, ne, ne podpira ga), potem ga ne moremo dodati. Z metodami lahko dajemo izrazom *pomen*, ne moremo pa spreminjati *sintakse jezika*. Ne moremo si izmisliti novega tipa oklepajev.

Prav tako ne moremo, recimo, spreminjati prioritete operatorjev, saj se leta razrešuje v trenutku, ko Python bere program (prebere ga v "abstraktno sintaktično drevo", ast), na to pa ne moremo vplivati.

Kakšne metode še obstajajo?

Kaj vse je še mogoče definirati - tako kot `__add__` in `__str__`? Seznam je še dolg. Na njem ni kaj prida takšnega, kar bi bilo smiselno uporabiti za vektorje. Vseeno pa naštejmo nekaj zanimivih.

Metode `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, `__ge__` skrbijo za primerjanje (less than, less or equal, equal, not equal greater than, greater or equal). Vektorjev ne moremo primerjati po velikosti, tako da teh metod tule nima smisla definirati.

Z `__or__`, `__and__` in `__not__` lahko sestavimo logične operacije, kadar je to smiselno. Pri vektorjih - ni.

Če damo razredu metodo `__call__`, bo objekte tega razreda mogoče poklicati. Če mu damo `__iter__` bo čez objekte mogoče pognati zanko `for`. No, ko smo že ravno pri tem:

```
for coord in v:
    print(coord)
```

```
3
4
13
```

Kako to deluje? Python preveri, ali ima `Vector` metodo `iter`. Če bi jo imel, bi morala vrniti iterator, nekaj z metodo `__next__` (ki je tisto, kar se v resnici kliče, ko pokličemo funkcijo `next`). Ker je nima, pa poskusi najti kakšen obvoz. Naš `Vector` ima `__getitem__` in ta sprejema `int`. To zadošča: zanka `for` pobere elemente z indeksi 0, 1, 2, 3, 4, ... in tako naprej, dokler `__getitem__` ne vrne napaka `IndexError`.

Seznam posebnih metod je še dolg. Za takšne stvari ni potrebno, da jih človek zna na pamet. Spoznali pa smo osnovni princip. Zdaj, ko vemo, kako stvari delujejo, bomo takrat, ko bomo kaj potrebovali, le pogledali v seznam posebnih metod.

Izpeljevanje iz vdelanih tipov

Če se komu zdi, da je `Vector` precej podoben seznamu, mu bom pritrdil. V resnici je `Vector` videti kot seznam, ki ima nekoliko drugačno seštevanje in ki mu je mogoče izračunati absolutno vrednost. Če je tako, pa izpeljimo `Vector` iz `list`!

```
from math import sqrt

class Vector(list):
    def __abs__(self):
        from math import sqrt
        return sqrt(sum(x ** 2 for x in self))

    def __add__(self, other):
        return Vector(*(x + y for x, y in zip(self, other)))

    def __mul__(self, other):
        if isinstance(other, Vector):
```

```
            return sum(x * y for x, y in zip(self, other))
        else:
            return Vector(*(x * other for x in self))

    __rmul__ = __mul__

    def __neg__(self):
        return -1 * self

    def __sub__(self, other):
        return self + -other

    def __truediv__(self, other):
        return self * (1 / other)
```

Definicija razreda je zdaj malo krajša, saj smo podedovali kup stvari od seznama. Vrednosti po novem niso več v `coords`, temveč kar v objektu samem - v `self`, če hočete. Konstruktor že imamo, prav tako izpis (se bomo pač zadovoljili z oglatimi oklepaji) in dolžino. Povoziti moramo le še aritmetične operacije.

Obenem pa je `Vector` podedoval še vse, kar ima seznam, recimo `append`. Ali je to pametno ali ne pa ... No, najbrž ni. :)